

Separating JavaScript Applications by Processes

Zoltán Horváth, Renáta Hodován, and Ákos Kiss

Department of Software Engineering, University of Szeged
Honvéd tér 6., H-6720 Szeged, Hungary
{hzoltan, akiss}@inf.u-szeged.hu, hodovan.renata@stud.u-szeged.hu

Abstract. JavaScript-enabled web browsers are one of the most important application platforms of today. However, web browsers are not perfect; sometimes they freeze and crash. Some new browsers take the approach of starting a new process for each tab and window. However, the question remains open: Is there a price to pay? In this research-initiating paper, we investigate the differences between running JavaScript programs simultaneously in separate processes and running them in the same runtime environment. Our experiments lead to conclusions for single-core and dual-core X86 architectures, but raise several new questions and drive the set-up of a research agenda.

1 Introduction

JavaScript-enabled web browsers are one of the most important application platforms of today. People use them in offices and at home, for reading news, sending emails, playing games, and purchasing goods – just to name a few common activities. Moreover, they are used on mobile phones as well, and not solely for surfing the web, but as extension platforms, too. However, web browsers are not perfect; sometimes they freeze, sometimes they run out of memory and crash. If one window or tab crashes, most of the time it kills all the other open tabs and windows as well. If a widget crashes on a phone, the effect may propagate to all the other JavaScript-based applications, too. This may leave unsatisfied users behind.

To avoid bad user experience – and also to lower the security risk of web browsing by utilizing the operating system-level separation of data –, some new browsers, like Google Chrome [1], take the approach of starting a new process for each tab and window. This seems to be a safe and sound approach, but the question remains open: Is there a price to pay? Is there a memory consumption or performance overhead for a per-process solution? Is this approach generic and valid for both desktop and mobile applications, or is it inappropriate for mobile devices due to their resource constraints?

In this research-initiating paper, we try to give answers to some of these questions. We investigate the differences between running JavaScript widgets simultaneously in separate runtime environments (i.e., in separate processes) and running them in the same runtime environment.

In the following parts of the paper, we first describe our measurement setup in Section 2, and give the results of our experiments in Section 3. Section 4 gives a summary of the paper and points out directions for future work.

2 Measurement Setup

In our experiments, we have chosen the WebKit [2] project to work with, which is the basis for both the aforementioned Google Chrome [1] web browser and the web runtime (WRT) [3] of the popular S60-based mobile phones. The project is in constant development, we used revision r43208 for the measurements. We have built WebKit for x86-linux target, linked to the Qt 4.5.1 libraries. As the measurement hardware, we used a dual-core CPU (sometimes with one of the cores disabled) clocked at 1.86 GHz, with 1.5 GB RAM and swap disabled.

Usually, measuring the runtime of processes is an easily solvable problem, but producing meaningful figures on memory consumption is not that trivial, especially in the presence of multiple processes and shared memory. Simply summing up the figures reported by tools like ‘ps’ can be misleading [4]. Shared memory segments (e.g., from dynamically linked libraries) can appear several times in the sum, thus giving invalid results.

Fortunately, Linux’s 2.6.14 kernel [5] adds support for smaps to the proc file system. Smaps files give a detailed view into the internals of the kernel, on the address space, memory region mappings and access permissions on a per-process basis. This information does not only allow us to exactly determine the memory consumption of a process (or of several processes) but it makes more detailed analysis possible. With the help of smaps files, we can separate code and data segments, shared and private regions, and we can do this even at library-level granularity.

For the sake of our experiments, we developed an execution and measurement script framework, utilizing the Linux::Smaps perl module [6]. In this framework, we sample the monitored processes every 0.5 seconds and log the results. In the log, we use the following granularity: we measure the summed size of the code and data sections of WebKit itself, the memory consumption of the Qt-related libraries, and we aggregate the consumption of all the remaining libraries (e.g., that of ‘libc.so’.) Moreover, we log the stack and heap usage separately as well.

3 Experimental Results

In our experiments, we investigated 3 popular JavaScript benchmark suites: SunSpider [7], V8 [8], and WindScorpion [9]. SunSpider is the official benchmark suite of WebKit, which contains 26 JavaScript files that try to represent real performance problems. The V8 benchmark suite contains 5 pure JavaScript benchmarks that Google Chrome developers have used to tune their V8 JavaScript engine, and finally, WindScorpion benchmark is a collection of 15 freely available real-life JavaScript examples.

First, Figure 1 shows some basic information about the used benchmarks. Clearly, the WindScorpion benchmark has the longest runtime and also has the highest memory consumption. SunSpider takes about half the time to finish, with somewhat less memory consumption, and V8 runs quarter of WindScorpion’s time with memory consumption similar to that of SunSpider. From the figure it is also visible that the core of WebKit and the Qt libraries consume roughly the same amount of memory, and all the remaining libraries take up memory on the same scale. Stack usage is negligible, compared to other components, while the heap is responsible for the largest part of the memory usage.

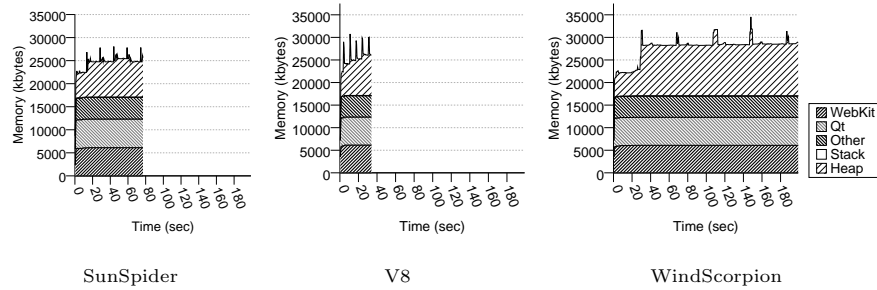


Fig. 1. The execution time and memory consumption of three benchmarks suites.

The next three figures (Figs. 2, 3, and 4) form the main contribution of this paper. In those figures, we present how the execution time and the memory consumption of the benchmarks change when two instances of them are executed parallelly. For all benchmarks, we performed measurements with both a dual-core and a single-core setup (in the latter case, the measurements were carried out on the same hardware but Linux was instructed to disable one of the cores), and with the two widgets executed in the same process (but in separate windows) and in two separate processes.

Interestingly, both for SunSpider (Fig. 2) and V8 (Fig. 3), executing the two widgets in the same process on a single-core CPU yields both the shortest runtime and the lowest memory consumption. Moreover, for SunSpider, both on single-core and dual-core setups, the same-process execution mode wins over the separate-processes mode, both in terms of performance (14-21%) and memory usage (12-23%). The same holds for the V8 benchmark when executed on a single-core setup; i.e., the execution is 12% faster and consumes 25% less memory in a single process. (On dual-core, the execution time difference between the same-process and separate-processes approaches vanishes.)

The measurements with the WindScorpion benchmark (Fig. 4) show a quite different picture. In that case, while the single-core setup follows the trends that we have seen with SunSpider and V8, the dual-core setup clearly benefits from the separate-processes execution mode. The execution time in separate-processes

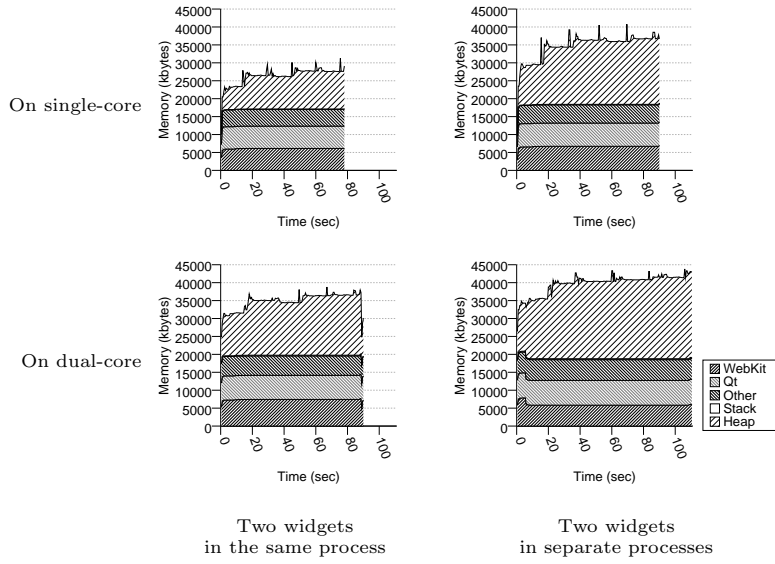


Fig. 2. The execution time and memory consumption of two parallelly executed SunSpider benchmark widgets.

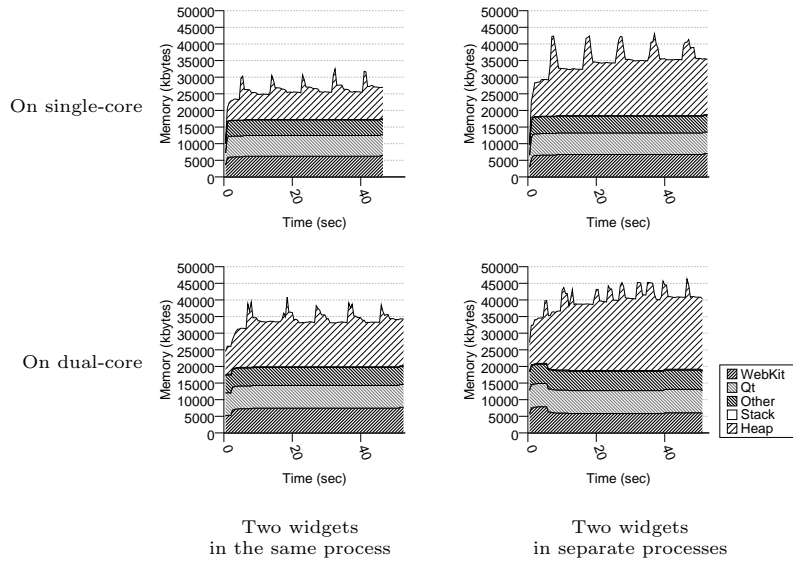


Fig. 3. The execution time and memory consumption of two parallelly executed V8 benchmark widgets.

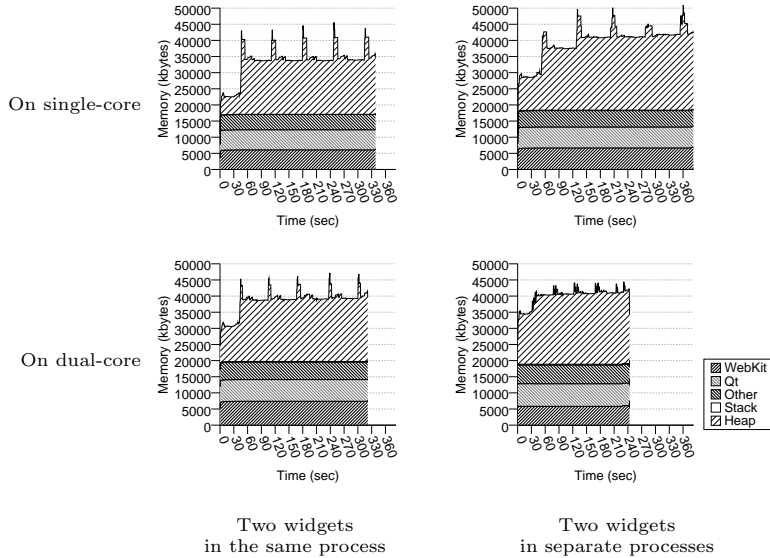


Fig. 4. The execution time and memory consumption of two parallelly executed WindScorpion benchmark widgets.

mode falls by one-fourth, compared to the same-process mode, while the average memory usage increases only slightly and the peak memory usage remains about the same.

The rationale behind the differences in the behaviour of the SunSpider-V8 benchmarks and the WindScorpion benchmark might be that SunSpider and V8 are artificial micro benchmarks composed of quick JavaScript programs, while WindScorpion consists of long-running programs that are closer to real-life applications. However, for now, that is only speculation. What we can state is that on single-core setups, the separate-processes execution is always worse both in terms of execution time and memory consumption than the same-process mode. For dual-core setups, long-running JavaScript programs might benefit from being executed in separate processes.

4 Summary and Future Work

In this paper, we investigated how can two JavaScript programs be parallelly executed in the most efficient way, both in terms of overall performance and memory consumption. Our experiments revealed that on single-core hardware setups, running two JavaScript programs in the same execution environment (i.e., in the same browser or web runtime) is always a win compared to running them in separate processes. On dual-core architectures, the picture is not that clear. Our measurements suggest that on dual-core setups, executing long-

running JavaScript programs in separate processes might result in a performance gain with no remarkable increase in memory usage.

Our intention is to make this paper a starting point of future researches. During our experiments, several questions arose, that set up a research agenda:

- What is the effect of executing more than two JavaScript applications parallelly?
- Do the measurements change considerably if the investigated widgets contain complex web pages, not only performance benchmark scripts?
- How do different memory allocators affect the runtime and the memory consumption?
- Are the conclusions general, valid for several architectures (e.g., for ARM), or are they specific for X86 only?
- Which execution mode is appropriate for desktop applications and which one is applicable to mobile devices?
- Do the conclusions hold for different types of execution engines (e.g., interpreter-based and JIT technology-based ones) and different web browsers?

As we will proceed along this agenda, the results will be hopefully useful for web runtime designers. We hope that the conclusions will help them balancing between security-stability and performance-memory consumption.

References

1. Google, Inc.: Chrome – version 2.0.172.33
<http://www.google.com/chrome> (Accessed 1 July 2009).
2. Apple, Inc.: The WebKit open source project
<http://webkit.org/> (Accessed 1 July 2009).
3. Nokia, Inc.: Web runtime quickstart
http://www.forum.nokia.com/Technology_Topics/Web_Technologies/Web_Runtime/QuickStart.xhtml (Accessed 1 July 2009).
4. Devin: Understanding memory usage on Linux (February 2006)
<http://virtualthreads.blogspot.com/2006/02/understanding-memory-usage-on-linux.html> (Accessed 1 July 2009).
5. Linux Kernel Organization, Inc.: The linux kernel archives
<http://kernel.org/> (Accessed 1 July 2009).
6. Foertsch, T.: Linux::Smaps perl module – version 0.06
<http://search.cpan.org/~opi/Linux-Smaps-0.06/lib/Linux/Smaps.pm> (Accessed 1 July 2009).
7. Apple, Inc.: SunSpider JavaScript benchmark
<http://www2.webkit.org/perf/sunspider-0.9/sunspider.html> (Accessed 1 July 2009).
8. Google, Inc.: V8 benchmark suite – version 1
<http://v8.googlecode.com/svn/data/benchmarks/v1/run.html> (Accessed 1 July 2009).
9. Department of Software Engineering, University of Szeged: WindScorpion
<http://www.sed.hu/webkit/?page=downloads> (Accessed 1 July 2009).